Chapter 5

Control of Software

5.1 Overview

This chapter addresses how a user can control and see what GrIDS software is executing where in the distributed GrIDS system. This chapter builds on the mechanisms described in the data source library chapter (chapter 4), and is also closely tied in with the organizational hierarchy (chapter 6). In particular, the reader will need a sense of an organization as a tree of departments, as described in that chapter, and also in the Introduction (chapter 1).

When the user wants to see what software is executing where, the GUI shows her a structured list of hosts and subdepartments in the department, along with what software is executing on the host/subdepartment and what software it *can* run but is not currently running with options to turn on or off the software.

If the user chooses to do so, then a request is sent to the Module Controller on the host, which then starts up or turns off the appropriate software. It is the process by which such requests are handled that is described in this chapter.

The software that the user sees and controls through these mechanisms includes:

- graph aggregators
- GrIDS monitoring devices i.e. sniffers, TCP wrappers
- point IDSs
- Departmental Software Managers (described below)

However, we use the same mechanisms to mediate access to *any* variables that are settable via the data source library mechanisms of chapter 4.

5.2 Software Managers

A software manager runs for each department, and handles requests originating from the interfaces. These requests involve starting up and shutting down software that is executing, returning lists of what software is and is not running, and changing parameters of running software.

The software manager also keeps track of hosts that are currently not responding to GrIDS messages. It knows not to include them in actions that are taken (to avoid delays while the protocols time out), but it probes them on a regular basis to see if they have come up again.

Software managers listen on a given port (which the organizational hierarchy software knows about) for requests. Note that there may be more than one software manager running on a given host if more than one domain is "headquartered" there, so a single global port cannot be used.

5.3 Access Control

Certain transactions are initiated by a User Interface in direct conversation with a Software Manager. In such cases, the Software Manager mediates *human* access control.

In other cases, a Module Controller may be contacted by some entity claiming to be a Software Manager, on behalf of some unknown user. To ensure that such transactions are authorized, the Module Controller performs *machine* access control, based on the Host:Port of the alleged Software Manager from whom it received the command.

5.3.1 User Access Control

The software manager for a department maintains a *human* access control list, specifying the operations that can be performed on its hosts and subdepartments by particular users. (The initial implementation will *not* support explicit *revocation* of some access capability.) Each entry of the list is a pair as follows,

$$(user\_id, Operation)$$

User_id uniquely identifies a GrIDS user. *Operation* is the capability either to Read or to Write State Variables everywhere within the subtree rooted at that department.

The access control list of a software manager inherits the access control lists from all its ancestor software managers in the "Software Manager Hierarchy". Hence, the access control list of a software manger consists of two parts: the inherited access control list (i_acl), and the local access control list (l_acl). The relationship between the access control lists of a software manager (P) and its immediate child software manager (C) is shown as follows,

$$acl(C) = i\_acl(P) \cup l\_acl(P) \cup l\_acl(C)$$

Any change to the *acl* of a software manager is propagated to each of the *i_acl* of the descending software mangers. For example, removing an entry in the *l_acl* of a software manager results in removing that entry in the *i_acl* of all the descending software managers ... *unless* that entry also was inherited by the software manager from whose *l_acl* it was just removed. (In this scheme, we do not allow children to revoke capabilities they have inherited from their ancestors.)

5.3.2 Machine Access Control

(This section is replicated below.)

Most commands are intended for "ordinary" Data Source modules. A Module Controller will allow only its own parent Software Manager to access (via GET or SET) StateVars on an ordinary module. But 2 types of special modules require different access control.

If a module is a Software Manager or an Aggregator, then it may be running on behalf of another department – ie, not the dept on whose host it is headquartered. In that case, an Aggregator should trust commands from the Host:Port of its own dept's Software Manager, and a Software Manager should trust commands from the Host:Port of its own Software Manager parent. In general, these may be different from the local Module Controller's Software Manager parent.

Thus, each instance of an Aggregator or Software Manager must be associated with an ACL specifying which Host:Port is allowed to reconfigure or shut it down. (The initial implementation will allow an ACL to contain only a single Host:Port. This should be sufficient.)

The Module Controller will check that ACL before allowing Write access to an Aggregator. The Module Controller's own Software Manager parent is *implicitly* granted Read access to every module running on that host.

5.4 Trust Issues for Software Managers

Each Software Manager has its own self-settable StateVars (ie, *not* accessible by the Module Controller):

- parent Software Manager (Host:Port)
- child Software Managers (list of Host:Port)
- child Module Controllers (list of Host:Port)
- (Host:Port) of Aggregator for our dept
- (Host:Port) of Module Controller for the above Aggregator (might not be the same port on every host)
- Inherited ACL (human IDs)
- Organizational Hierarchy Server (Host:Port)
- its own department ID (unique)
- Local ACL (human IDs)
- timeout period (to wait for kids to reply to pings it issues to implement a "host_status" command)
- list of "pending" transactions it should attempt to complete (Typically, these involve some of its *former* children. Therefore it should explicitly store the relevant Host:Ports here.)

A local config file contains those last 5 items for persistent storage. On wakeup, a Software Manager needs to confirm its place in the Organizational Hierarchy, and update any cached info that may be stale. It should not risk passing on any stale info to what may or may not be its children (hence its config file only stores those last 5 items), nor should it waste bandwidth by sending "unbuffered" info.

Therefore, a new Software Manager first contacts the Organizational Hierarchy Server and presents its department ID. If the Software Manager has an invalid department ID, or if somehow that department already has a different Software Manager, then the Organizational Hierarchy Server tells the new Software Manager to die. The Organizational Hierarchy Server should make a note of strange incidents like this in its own log file.

Otherwise, the Organizational Hierarchy Server tells the new Software Manager its:

- parent Software Manager (Host:Port)
- child Software Managers (list of Host:Port)
- child Module Controllers (list of Host:Port)
- (Host:Port) of Aggregator for our dept
- (Host:Port) of Module Controller for the above Aggregator

The Software Manager then waits (this timeout should be implemented as a separate State Variable) to be contacted by its parent, from whom it receives a single catenation of its ancestors':

- Inherited ACL (human IDs)

It then contacts each child Software Manager, and propagates both Local and Inherited ACLs. If some child Software Managers do not respond soon enough (a different timeout State Variable), it marks those child Software Managers as presumed down, and appends those transactions to its "pending" file.

Note that upon wakeup, before a Software Manager can connect to the Organizational Hierarchy Server, it might receive a connection from someone claiming to be its parent. In this case, the Software Manager should accept an Inherited

ACL from the alleged parent. However, it should *not* pass that data on to its own children until it can connect to the Organizational Hierarchy Server and confirm the identity of its alleged parent.

Next, the Software Manager performs an uptime/liveness check of its child hosts. At some safe place in its main loop, the Software Manager checks whether it has recently (yet another timeout Variable) polled the Module Controller of each child to see if it is alive. If appropriate, it polls them again. It then attempts to process (ONE OF, OR ALL?) the "pending" transactions stored in its file, if any of the intended recipients are currently alive.

SCHEDULING ISSUES: PRIORITY, FAIRNESS, INTERACTIVE RESPONSE ???

5.5 Trust Issues for Module Controllers

(Much of this section should be replicated in the chapter on Data Sources.)

At startup, the Module Controller reads its own config files to determine:

- parent Software Manager (Host:Port)
- Aggregator to whom Data Source Modules report (Host:Port of Aggregator itself)
- Organizational Hierarchy Server (Host:Port)
- timeout period (to wait for local modules to respond to reconfigs)

The uptime/liveness protocol between a Module Controller and its parent Software Manager may be initiated by either party. Each party should identify itself, and assert its kinship relationship with the other. This protocol may cause a Module Controller to question its parentage (rightly or wrongly). If a Module Controller believes an unauthorized Software Manager parent is trying to claim it, or if its overtures are rejected by its purported parent, then the Module Controller should contact the Organizational Hierarchy Server. This method will handle cases where a host (and its Module Controller) is down, and misses the notification that it is being moved to a different department (Software Manager), and henceforth should obey commands from a different parent Software Manager.

The Module Controller has 2 special config files: One contains commands to invoke (potentially multiple) instances of certain programs, with appropriate parameters (some of those parms may be variables, e.g., all dumb Data Sources must be given the Host:Port of this department's Aggregator). This config file is writable by the Module Controller, since it represents a dynamically changing sequence of tasks the Module Controller should execute each time it awakens.

The Module Controller can read – but not write – its second special config file. This file contains *patterns* that *constrain* what can appear in the first special config file. In this manner, we can ensure that pathnames and parameters in the first config file comprise "safe" commands, yet also allow the Module Controller to dynamically invoke new instances of programs.

(The Module Controller also needs a file to store the PIDs corresponding to running GrIDS programs. It is not sufficient to store these in RAM, because whenever the Module Controller awakens, it must ascertain whether some of its children are still alive, to avoid invoking redundant instances of those tasks. This file should also store the names/locations of the config files corresponding to those tasks, and the department ID that "owns" each task.)

Initially, a Module Controller might startup both Aggregators and Software Managers, and each will read config info from its own file(s). From its file(s) or parms, an Aggregator will learn the Host:Port of its parent Aggregator, and of its departmental Software Manager. A Software Manager will learn (via config files or parms) the ID of the department it represents, and will contact the Organizational Hierarchy Server to learn the Host:Port of its parent Software Manager.

Thus, initially, various Module Controllers will correctly startup all Software Managers via this method.

The main loop of the Module Controller receives a TCP connection, decodes the incoming command, decides whether to allow access (based on the source Host:Port), processes the command, responds to the sender, and closes the connection.

Most commands are intended for "ordinary" Data Source modules. A Module Controller will allow only its own parent Software Manager to access (via GET or SET) StateVars on an ordinary module. But 2 types of special modules require different access control.

If a module is a Software Manager or an Aggregator, then it may be running on behalf of another department – ie, not the dept on whose host it is headquartered. In that case, an Aggregator should trust commands from the Host:Port of its own dept's Software Manager, and a Software Manager should trust commands from the Host:Port of its own Software Manager parent. In general, these may be different from the local Module Controller's Software Manager parent.

Thus, each instance of an Aggregator or Software Manager must be associated with an ACL specifying which Host:Port is allowed to reconfigure or shut it down. (The initial implementation will allow an ACL to contain only a single Host:Port. This should be sufficient.)

The Module Controller will check that ACL before allowing Write access to an Aggregator. The Module Controller's own Software Manager parent is *implicitly* granted Read access to every module running on that host.

It is the responsibility of a Module Controller to perform this enhanced access-control checking when it receives commands directed at an Aggregator module. Thus, to perform *authorized* reconfiguration of a resident-alien Aggregator, the Module Controller must be able to read (and write) a file containing the Aggregator's ACL.

In contrast, since a Software Manager will reconfigure itself, it performs the access-control check itself (the Module Controller is *not* involved in the protocol at all). Although the local Module Controller does not need to read a resident-alien Software Manager's ACL file, it *does* need read access to certain info in the Software Manager's config file.

This is because a Module Controller's own Software Manager parent has the "right" to inventory all modules running within its domain. Hence the Module Controller needs *read* access to config files of resident-alien Aggregators and Software Managers, so it can determine who they represent. (This capability allows a Module Controller to return a helpful error message if a User Interface tries to delete its host, yet a foreign Software Manager is headquartered on it.) Note that a Module Controller's own Software Manager parent is not allowed *write* or *shutdown* access to a resident-alien Aggregator or Software Manager that is already running.

[This schema allows inconsistent reads under certain race conditions, where a Module Controller is reading a resident-alien Software Manager's configuration while that Software Manager is reconfiguring itself via its own TCP connection. We believe this problem is minor, and can be ignored initially.]

Finally, who should be authorized to start a new Software Manager or Aggregator dynamically? A Module Controller should obey incoming commands from its *own* departmental Software Manager to start a *new* Aggregator or Software Manager. (These commands will be SETs directed at the Module Controller itself, to which its parent Software Manager implicitly has Write access. These commands will specify the Host:Port for future access-control to that newly started special module.) But after launch, write-access to a new Aggregator or Software Manager should be allowed only from whatever Host:Port was specified in its startup command (and stored in its ACL file). In this manner, a native Software Manager authorizes the residency of a foreign module. (By setting strict constraints on the invocation parameters of a Software Manager in the special config file, a department can prevent certain foreign departments from headquartering their Software Managers or Aggregators on its host.)

5.6 IMPLICATIONS for GET/SET FORMAT

Because *multiple* instances of the "same" program (Aggregator or Software Manager) may be running on the same host, how do we tell the Module Controller *which* instance of a module to read or reconfigure?

This situation could arise for multiple Aggregators (GET and SET), and for multiple Software Managers (GET only; their reconfiguration does not involve the local Module Controller.)

We *could* continue using the previous GET/SET format: SET-ting an Aggregator could distinguish between multiple Aggregators via the HOST:PORT that initiated the reconfig command. If the ACL for an Aggregator only has a *single* value, then this method will work ok. GET commands could be applied to *all* instances of the relevant module (either Aggregators or Software managers).

Alternatively, the GET/SET command could specify the port or the *home department ID* of a particular multi-instance module. The Module Controller would need to learn (presumably from the modules' config files) which port or *home department ID* corresponded to which instance (PID) of a multi-instance module.

5.7 IMPLICATIONS for Ruleset Updating

For Software Manager of dept C to update C's Aggregator, it is not sufficient to know the Aggregator's Host:Port (to which GrIDS reports are sent by C's children). Instead, to SET StateVars on C's Aggregator, C's Software Manager needs to know the Port for the Module Controller at which C's Aggregator resides. (Generally this Port will be the same on all hosts ... but not necessarily, especially as we scale up!)

The Organizational Hierarchy schema already requires that for each host, it stores the Port of its Module Controller. Moreover, the initial wakeup procedure for a Software Manager specifies that one item it will receive from the Organizational Hierarchy Server is the Host:Port of the Module Controller that controls its Aggregator.

Since presumably Rulesets change more frequently than the Organizational Hierarchy, the Software Manager should *cache* that Host:Port, from its most recent dialog with the Organizational Hierarchy Server.

Humans will attempt to ensure that all Module Controllers occupy the *same* known port on all hosts. This value will be loaded by hand into each host node in the Organizational Hierarchy file. When a Module Controller must occupy a non-standard port, it is the responsibility of human operators manually to update the Organizational Hierarchy file to reflect that situation.

It is *possible* (though unlikely) that a Module Controller may change Ports, eg, perhaps its host dies, and upon restart, it awakens to find that its previous port has been taken by some other process. Any Software Manager needing to contact that Module Controller must *assume* it resides on the previously-cached port, and send its message there. (The node for a host at the Organizational Hierarchy server does not carry info about which Aggregators may be running on that host. So unless someone does an exhaustive search after a human manually updates the port of a Module Controller, the Organizational Hierarchy Server cannot know to update a Software Manager if/when its Aggregator's Module Controller changes Ports.)

If the Software Manager fails to receive an ACK from the Module Controller before the timeout, then the Software Manager should contact the Organizational Hierarchy server and ask for the correct (changed) Port for the Module Controller at that host.

5.8 Deleting Hosts Running Critical Modules

When a host is to be deleted, we already specified this operation should fail (with an informative error message) if the host is running any Aggregator or Software Manager. Thus, the Module Controller must know every GrIDS module it is running, including resident-aliens. Because the Module Controller has *implicit* Read-access to resident-aliens' config files, this allows the Module Controller to inform a User Interface *whose* Aggregator is causing an error in response to a host-delete command.

[redacted]

5.9 Software Control Protocol

All of these packets use the GCPF described in chapter 3. The reader will also need to be familiar with the get/set protocol described in chapter 4. All messages here use the field separation scheme detailed there.

5.9.1 Set Host Variable (header *shv*)

This message tells the software manager to set a particular variable on a particular host. The host must be a member of the associated department. The fields are:

- Username
- Password
- Host Name
- Data Source Name
- Department ID on whose behalf the Data Source is running
- StateVarName
- Value
- StateVarName
- Value
- StateVarName
- Value
- StateVarName

5.9.2 Get Host Variable (header *ghv*)

This message tells the software manager to return a particular variable on a particular host. The fields are:

- Username
- Password
- Host Name
- Data Source Name
- Department ID on whose behalf the Data Source is running
- StateVarName
- Value
- StateVarName
- Value
- StateVarName
- Value
- StateVarName

5.9.3 Set Dept Variable (header *sdv*)

This message is used to tell the software manager to set variables that are handled in particular ways and need to be updated across the whole department, such as rulesets and access control variables. The fields are:

- Username
- Password
- StateVarName
- Value
- StateVarName
- Value
- StateVarName
- Value
- StateVarName

5.9.4 Get Dept Variable (header *gdv*)

This message is used to get summary information from the software manager. For the specified variables, the returned information will be a list of host-variable-value tuples which give the value of the variable on every host in the department. A null value for a variable name indicates that host is presently down.

- Username
- Password
- StateVarName
- Value
- StateVarName
- Value
- StateVarName
- Value
- StateVarName

Chapter 6

The Organizational Hierarchy

6.1 Introduction

The organization is assumed to be divided up in a hierarchical tree structure. Each internal node corresponds to a department in the organization. Each leaf node corresponds to a host/machine. Hosts are referred to by their fully qualified DNS names, while departments have unique identifier strings.

Each department has an associated graph engine, and a software manager. The engines use the departmental hierarchy as the means for aggregation of graphs, as described in chapter 2. The software managers control software operation in their particular department—their operation is detailed in chapter 5.

This chapter deals in how the information about the organizational hierarchy is stored and how it can be dynamically changed while staying in a consistent state.

6.2 An example

In order to make the overall scheme clearer to the reader, we will step through a complete example of a transaction on the hierarchy, showing how all the major elements of the system are affected. In subsequent sections, protocol details will be described.

Our scenario is depicted in figure 6.1. We begin with the start up of the interface component. At the outset, the user must supply the interface with the name of a department to which he believes he has access, his user identifier, and his credentials. In this case, let us suppose that he has access at department C, but not at B or A. Because of his access at C, he will automatically have access in the subtree below this.

In the following, we use the notation S_C to refer to the software manager at C, A_C to refer to the aggregator at C, M_C to refer to the module controller on the machine on which S_C and A_C are running, and similarly for the other departments. The organizational hierarchy server is O, and the interface is I.

I contacts O to request a copy of the hierarchy below C. O contacts S_C first and verifies that the user does have access at C and his credentials match up. S_C replies in the affirmative. Then O supplies I with a copy of the hierarchy rooted at C. I displays this on the user's screen. The copy is marked with

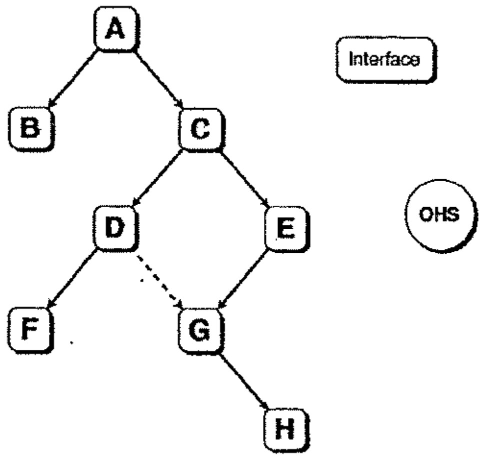


Figure 6.1: An example hierarchy. Department G is about to be moved from under department E to under department D. An interface and the organizational hierarchy server are also shown.

a *version number* which can be used to determine if it is still valid. O also marks down that I has a copy of this part of the hierarchy. If someone else were to change the hierarchy, O would send a message to I telling it that its version was out-of-date.

Suppose that, having inspected the hierarchy, the user decides to move department G (and by implication, its subtree also) to be under D instead of E. The first step is to send a message to O. This message describes the planned action and supplies the hierarchy version number on which the planned action was based. O first determines if the planned action is consistent with existing locks in the hierarchy and is based on an up to date version of the hierarchy. If so, it locks the hierarchy appropriately. Next, it contacts S_D and S_E to verify that the user has appropriate permissions. If not, it releases the locks. If any step has failed, an appropriate error message is supplied to the user at this time. Assuming that the action appears feasible, O gives permission for I to go ahead.

Now I contacts S_E and informs it that G is to be moved. Assuming that S_E believes M_G, S_G and A_G to be up, it sends messages to S_G telling it that all rulesets inherited from E are to be deleted. S_E also sets these rulesuts null via M_G. S_G does the same recursively for its children, and then acknowledges to S_E. Then S_E sends set messages to M_G which alter where S_G and A_G send messages and where M_G believes its parent to be. Once these actions have succeeded, S_E updates its own data structures and acknowledges completion to I.

Next I contacts S_D and informs it of the change. S_D then performs the following actions.

- Contacts S_G and M_G to give them the new rulesets.
- Sets the new value of the inherited access control list on S_G which recursively does the same for the tree below it.
- Sets default policy variables via M_G and S_G, with S_G to do the same below it.
- Acknowledges success to I

When this is complete, I reports back to O that the action is complete. O then removes appropriate locks on the hierarchy. Finally, O contacts any interfaces that might have been affected and tells them that their information is invalid.

Note that this procedure causes problems if any of the critical actors dies in the middle.

6.3 The Organizational Hierarchy Server

The organizational hierarchy server is designed so that it can be shut down and then restarted without any change in the operation of the hierarchy (though no transactions or new viewing will be possible during the shutdown). Transactions in progress before the shutdown must be able to run to completion after the shutdown. Hence, it is necessary for the server to store its state on shutdown and read it again on startup.

The structure of the organizational hierarchy will be stored in a file, represented in modified DOT format. For each internal node in the organizational hierarchy, the file will contain:

- is node a department or a host (zero children is not a reliable indicator)
- department name (must enforce *uniqueness* of dept name)
- location of this department's aggregator (host:port)
- this node's parent department ID (unless it's the root node)
- hosts directly attached to this department (ID list)
- this node's child departments (ID list)
- location of Software Manager for this node (host:port) (this allows Software Managers to be sparsely distributed within the tree, vs. mapping 1-1 to departmental nodes)

For each host node in the organizational hierarchy, the file will contain:

- is node a department or a host (zero children is not a reliable indicator)
- host name
- this node's parent department name
- location (port) of Module Controller (omit if there is a universal port number for it)

The syntax of the file is as follows,

```
<Organizational-Hierarchy-Structure> ::= <node>

<node> ::= [<host-node>; | <dept-node>;] [<node>]

<host-node> ::= 'host, ' <host-name> ','
                <dept-name> ',' <MC-port>

<host-name> ::= <alphanumeric-string>

<dept-name> ::= <alphanumeric-string>

<MC-port> ::= <port-number>

<dept-node> ::= 'dept, ' <dept-name> ','
                <parent-dept-name> ',' <host-list>
                ',' <child-dept-name-list> ','
                <software-manager> ','
                <aggregator>

<parent-dept-name> ::= 'null' | <dept-name>
```

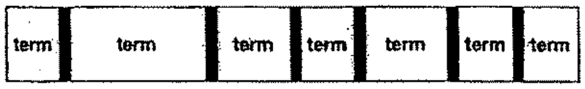


Figure 6.2: The hierarchy packet format. The separator, character 254, is shown in black.

```
<host-list> ::= '{' [<hosts>] '}'

<hosts> ::= <host-name> [',' <hosts>]

<child-dept-name-list> ::=
                 '{' [<child-dept-names>] '}'

<child-dept-names> ::= <dept-name>
                   [',' <child-dept-names>]

<software-manager> ::= <host-name> ':' <port-number>

<aggregator> ::= <host-name>  ':' <port-number>

<port-number> ::= <numerical-string>

<alphanumeric-string> ::= [A-Za-z0-9._]

<numerical-string> ::= [0-9]
```

We expect graph rule sets (and/or policies) to change fairly often. Thus we decided that rule sets will *not* be stored in this central Organizational Hierarchy file.

All accesses to this file will be channeled through an Organizational Server, to assure coherent updates and consistent views. This Server will constrain access to the file by implementing both read locks and write locks on the entire file. However, to permit finer locking granularity in the future, our protocol design will allow locks to be specified on particular departmental subtrees. If and when the Server becomes a bottleneck (e.g., when we scale beyond some point), then the Server implementation can be upgraded to allow more concurrent access, without modifying the protocol.

Also, note that Write locks on subtrees might be used to indicate that a Software Manager has "checked out" a particular subtree, thus distributing the org hierarchy for arbitrarily long periods. Making those checkout periods permanent might defuse various real-world departments' concerns about deployment and access control (e.g., they may not want others to read names of all their hosts).

6.4 Organizational Hierarchy Messages

6.4.1 Introduction

A number of messages are sent to the organizational hierarchy server, and they are described in this section. All these messages are GrIDS packets and use the GrIDS Common Packet Format (GCPF) defined in section 3.1.

All transactions follow a roughly similar pattern (a detailed description of the case of a move can be found in section 6.2). There is an initial request to change the hierarchy from an interface to the OHS. This has a header *htr* (hierarchy transaction request). The hierarchy server checks the feasibility of the request, and then responds with a message having a *htp* (hierarchy transaction permission) header which authorizes the interface to go ahead with the change. Alternatively, an *hve* message may indicate an error. After the change has been completed, the interface informs the hierarchy server of that fact with a message having a header of *htc* (hierarchy transaction completed).

In addition to transactions, there are also messages asking to view some portion of the hierarchy. These have a header *hvr* (hierarchy view request). The OHS will then respond with a *hv* (hierarchy view) message.

6.4.2 Packet Format

All hierarchy messages have a body which consists of a set of terms. Each term is separated from the next by a character 254. The terms themselves may consist of any character other than 254 and 255. There is no trailing 254. Only one hierarchy message can be contained in a GrIDS packet. The format is shown in figure 6.2.

As with the GCPF, the convention for storing these packets in files, or displaying packets to humans, is that the separator is followed by a newline.

6.4.3 Transaction Types

Hierarchy View Request (*hvr*)

These messages represent a request to see a particular section of the organizational hierarchy. The sender will generally be an interface, and the receiver will be the OHS. The messages have the following format (with one term per column of the table).

User	*Password*	*Dept*

User is the username of the individual on behalf of whom the interface is acting; *Password* was supplied by that user, and *Dept* is the name of the department for which the user wishes to view the hierarchy.

Hierarchy View (*hv*)

These messages are sent in response to a HVR, and supply the requested information. The format is as follows.

Dept	*Serial*	*Hierarchy*

Dept is the name of the department at which this particular hierarchy is rooted; *Serial* is a serial string for this information which can be used later to check validity of requests based on this information; *Hierarchy* is the requested view of the hierarchy—it is supplied in the hierarchy description language defined in section ??.

Two serial strings are considered the same iff they are identical considered as a sequence of ASCII characters.

Hierarchy View Error (*hve*)

These messages are sent in response to an *hvr* and indicate that the requested information cannot be supplied. The format is as follows.

User	*Dept*	*Error*

User is the requesting user; *Dept* is the name of the department at which this particular hierarchy is rooted; *Error* is an informative message which explains why the hierarchy is not being supplied. Typical reasons would be that the user didn't have permission, supplied an incorrect password, *etc.*

Note that *hvr, hv, and he* will all be carried over a TCP connection. Specifically, the interface opens a TCP connection to the OHS, passes the relevant packet, and then waits until the OHS supplies the appropriate response along the same connection before it closes the connection.

Hierarchy View Update (*hvu*)

These messages are initiated by the OHS to inform an interface that it now has invalid information. The messages are sent via UDP and are not acknowledged - this is only a best effort service. Ultimately, we rely on serial strings to catch that a request is based on old information. The format is as follows.

Dept	*Serial*

Dept is the name of the department at which this particular hierarchy is rooted; *Serial* is the serial string for the view which is no longer valid.

Hierarchy Transaction Request (*htr*)

There are a number of different kinds of these, depending on the particular transaction being attempted.

- *add_host*

  This message indicates that the requesting interface would like to add a host into the named department. The department must exist already, and the host may not already be part of the GrIDS system. The message format is:

Type	*Trans-id*	*User*	*Pass*	*Serial*
Dept	*Host*	*Port*		

User is the the requesting user-id, *Pass* her password, and *Serial* the view of the hierarchy on which this request was based. *Trans-id* is a transaction identifier string which will be used to refer to this particular transaction in future interchanges. It should be unique. *Type* is the string *add_host*, indicating the nature of the transaction. *Host* is the host to be added, and *port* is the port on which its module controller is running. *Dept* is the department name to which this host is to be added.

- *remove_host*

  This message indicates that the requesting interface would like to remove a host from the hierarchy. The host must currently be part of the GrIDS system. Note that this transaction cannot complete if any software managers or aggregators are running on the system, and the OHS should enforce this. The message format is:

Type	*Trans-id*	*User*	*Pass*	*Serial*	*Host*

User, *Pass*, and *Serial* have their usual meaning. *Type* is the string *remove_host*, indicating the nature of the transaction. *Host* is the host to be removed.

- *move_host*

  This message indicates that the requesting interface would like to move a host within the hierarchy. The host must currently be part of the GrIDS system. The message format is:

Type	*Trans-id*	*User*	*Pass*	*Serial*	*Dept*	*Host*

User, *Pass*, and *Serial* have their usual meaning. *Type* is the string *move_host*, indicating the nature of the transaction. *Host* is the host to be moved, and *Dept* is the department to move it to.

- *add_dept*

  This message indicates that the requesting interface would like to create a new department within the hierarchy. The department name must not already be in use. The new department will not have any children until they are added via separate commands. The message format is:

Type	*Trans-id*	*User*	*Pass*	*Serial*
Parent	*Dept*	$Host_S$	$Host_A$	

User, *Pass*, and *Serial* have their usual meaning. *Type* is the string *add_dept*, indicating the nature of the transaction. *Dept* is the name of the new department, and *Parent* is the department to make it a child of. $Host_S$ is the host on which to run software manager for the new department, while its graph engine will run on $Host_A$. This host must already be running a module controller—the port will already be known to the OHS.

- *move_dept*

  This message indicates that the requesting interface would like to move an existing department within the hierarchy. The subtree beneath it moves with it. Note that this causes no change in the physical location of the departmental facilities such as the software manager and the aggregator. It simply changes who they report to, inherit rulesets from, *etc.* The message format is:

Type	*Trans-id*	*User*	*Pass*	*Serial*
Parent	*Dept*			

  User, *Pass*, and *Serial* have their usual meaning. *Type* is the string *move_dept*, indicating the nature of the transaction. *Dept* is the name of the department to be moved, and *Parent* is the department to make it a child of.

- *new_root*

  This message indicates that the requesting interface would like to create a new root department for the hierarchy. This can only happen if there is no existing hierarchy and is intended just to be the mechanism for starting a hierarchy from scratch.

Type	*Trans-id*	*Dept*	
$Host_S$	$Port_S$	$Host_A$	$Port_A$

  Type is the string *new_root*, indicating the nature of the transaction. *Dept* is the name to give to the new department. $Host_S$ and $Host_A$ are the hosts on which to run the root department software manager and aggregator respectively. These must already be running module-controllers. However, the OHS must be told the ports of these, which is done in Port_S and Port_A respectively. Note that these are the ports of the *module-controllers* not the software manager and aggregator themselves. This is an oddity required in bootstrapping the system - we cannot add any hosts until we have a root department to add them into.

- *remove_dept*

  This message indicates that the requesting interface would like to delete a department within the hierarchy. The department must exist. All children of the department will also be deleted. The message format is:

Type	*Trans-id*	*User*	*Pass*	*Serial*	*Dept*

  User, *Pass*, and *Serial* have their usual meaning. *Type* is the string *remove_dept*, indicating the nature of the transaction. *Dept* is the name of the department to delete.

- *change_variable*

  This message indicates that the requesting interface would like to change some variables (perhaps rulesets or access control lists) in the subtree of some department. It is necessary for the organizational hierarchy server to be involved to ensure that the hierarchy is not changed during the process, possibly resulting in a corrupted state.

Type	*Trans-id*	*User*	*Pass*	*Serial*	*Dept*

  User, *Pass*, and *Serial* have their usual meaning. *Type* is the string *change_variable*, indicating the nature of the transaction. *Dept* is the name of the department at or below which changes will be made.

- *move_manager*

  This message indicates that the requesting interface would like to change the physical location of the software manager for some department, without changing the actual structure of the hierarchy.

Type	*Trans-id*	*User*	*Pass*
Serial	*Dept*	*Host*	

  User, *Pass*, *Serial*, and *Trans-id* have their usual meaning. *Type* is the string *move_manager*, indicating the nature of the transaction. *Dept* is the name of the department being changed, and *Host* is the name of the new host on which to locate the software manager. That host must already be part of the GrIDS system.

- *move_aggregator*

  This message indicates that the requesting interface would like to change the physical location of the aggregator for some department, without changing the actual structure of the hierarchy.

Type	*Trans-id*	*User*	*Pass*
Serial	*Dept*	*Host*	

  User, *Pass*, *Serial*, and *Trans-id* have their usual meaning. *Type* is the string *move_aggregator*, indicating the nature of the transaction. *Dept* is the name of the department being changed, and *Host* is the name of the new host on which to locate the aggregator. That host must already be part of the GrIDS system.

Hierarchy Transaction Error (*hte*)

These messages are used by the OHS to inform an interface that a transaction cannot be performed. The format is as follows.

Trans-id	*Error*

Trans-id is the identifier of the request in question, and *Type* is the type of transaction request. *Error* is an explanatory message.

Hierarchy Transaction Permission (*htp*)

These messages are used by the OHS to inform an interface that it may go ahead with a requested transaction. The format is as follows.

Trans-id

Trans-id is the identifier of the request in question.

Hierarchy Transaction Complete (*htc*)

These messages are sent to the OHS by an interface to say that a transaction has been completed and locks should now be released. In almost all cases, the format is as follows:

Type	*Trans-id*

Trans-id is the identifier of the request in question, and *Type* corresponds to the types in the *htr* messages.

However, there are also a few special cases in which additional information is present. This happens with the port numbers of software managers and aggregators. When a transaction involves moving one of these, the port number that will eventually be used cannot be known reliably at the outset of the transaction. Hence it must be supplied by the interface in the *htc* message at conclusion. The special cases are:

- *add_department*

Type	*Trans-id*	$Port_S$	$Port_A$

  Trans-id is the identifier of the request in question, and *Type* is the type of transaction request. $Port_S$ is the port number on which the software manager is now located, and $Port_A$ is the aggregator port.

- *new_root*

Type	*Trans-id*	$Port_S$	$Port_A$

  Trans-id is the identifier of the request in question, and *Type* is the type of transaction request. $Port_S$ is the port number on which the software manager is now located, and $Port_A$ is the aggregator port.

- *move_manager*

Type	*Trans-id*	$Port_S$

  Trans-id is the identifier of the request in question, and *Type* is the type of transaction request. $Port_S$ is the port number on which the software manager is now located.

- *move_aggregator*

Type	*Trans-id*	$Port_A$

  Trans-id is the identifier of the request in question, and *Type* is the type of transaction request. $Port_A$ is the port number on which the aggregator is now located.

6.5 The View Serial Number Mechanism

In the previous section, *hv* and *htr* messages have a *Serial* field. This section explains the significance and the management of that field.

The purpose of this mechanism is to ensure that users do not attempt, accidentally, to make transactions on the hierarchy when they have an inadequate view of it. This can happen either because their view is out of date, or because they are attempting transactions outside of any view they have obtained.

To prevent this, when the OHS gives out a view of some portion of the hierarchy in a *hv* message, it attaches a serial number. Subsequently, when an interface requests a transaction on the hierarchy, the OHS checks that the serial number is up to date. If not, the OHS gives an *hte* error message.

Chapter 7

The Network Monitor

In GrIDS, network connections are monitored using network sniffers. A sniffer examines raw data packets carried within the monitored network and reports the status of communication channels between system entities (users, hosts, programs, etc.) to its aggregator. The aggregator analyzes the reports and detects patterns of communication amongst system entities that indicate misuse.

7.1 Assumptions and Design Objectives

Only one aggregator per sniffer. All packets that the sniffer cannot recognize as part of a "session" or "connection" are dropped.

We will do our best to report the current state of the network. A design goal is not to miss packets during GrIDS operation and consequently omit connection reports.

7.2 Events

An event is an abstract network occurrence. An event may consist of a single network packet or a collection of packets, but that detail is hidden within the data source from the aggregator.

We describe some types of network connections that the sniffer will monitor.

A connection is characterized by a START event, a END, and several intervening stages (events). The intermediate stages varies according to the application protocol communicating via the connection. Regardless of protocol, the sniffer should report the START, END stages of a connection. The END event may be successful or unsuccessful (error).

Note that a long delay may exist between consecutive packets that belong to an event. In some of these cases, the sniffer will recognize the some but not all packets of the event and withhold reporting to the aggregator until after all relevant packets have been observed.

to report the beginning of an event in hopes of seeing the rest of the event puts the GrIDS system in significant danger of delaying or failing detection of something critical. For example, reporting a telnet connection should not be delayed until the connection is closed and all packets have been sniffed and considered. However, individual parts of a telnet connection which require multiple packets may be appropriately reported individually, rather than as multiple packet reports.

7.3 What to sniff for

The initial three packets corresponding to a TCP handshake are aggregated into a TCP_START event. This connection is uniquely labelled with the 6-tuple (src host, src port, dst host, dst port, seq, time) where host is the hostname, port is the numeric port identifier, seq is the initial sequence number of the SYN packet from the source to the destination host, and timestamp of the initial SYN packet with respect to the sniffer's local host clock.

An important connection attribute is the protocol type. The protocol type indicates the sniffer's best guess as to the type of application or data that is carried by this TCP connection. The types we anticipate identifying include TELNET, RLOGIN, RSH, HTTP, MOUNT, NFS, UNKNOWN, etc. The sniffer recognizes different connection types using the port number of the source and destination hosts. In some instances, the sniffer may examine the first few data bytes of the connection to guess the protocol type of connection (e.g., MOUNT, NFS). The protocol type attribute is a separate attribute from the port number attribute because some services (e.g., MOUNT) do not have a standard fixed port number associated with it.

Besides the above attributes, there are some protocol dependent attributes. For instance, for RLOGIN connections, there are attributes that describe the login name of the user on the client host, the login name of the user on the server host, the client-user's terminal type and the speed of the terminal.

The output of a sniffer includes reports of the following events: 1) Start of a TCP connection between hosts. 2) End of a TCP connection between hosts – normal close (FIN), a connection reset (RST), a timeout (TIM). 3) Start of a UDP session between a pair of hosts. 4) End of a UDP session between a pair of hosts. 5) ICMP messages.

Although UDP is not a connection oriented network protocol, agents that communicate via UDP often exchange a

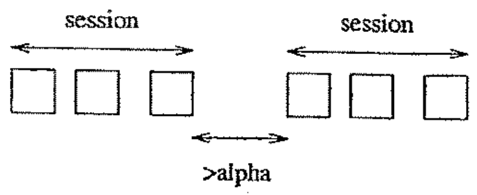


Figure 7.1: UDP sessions.

type	prot	stage	status	other attributes
TELNET	TCP	START	SUCC	
TELNET	TCP	OPTNEG	SUCC	TERM SIZE
TELNET	TCP	AUTH	SUCC	LOGIN PASSWORD
TELNET	TCP	AUTH	ERROR	LOGIN PASSWORD
TELNET	TCP	END	SUCC	
TELNET	TCP	END	ERROR	
TELNET	TCP	EXISTING	SUCC	

Table 7.1: TELNET events.

series of related UDP packets. We define a UDP session as the collection of UDP packets between a pair of hosts originating from the same port numbers. Packets that occur within a time window are part of the same session, and packets outside the window are treated as members of separate UDP sessions. A session's time window terminates when more than a certain amount of time, say α seconds, passes without a next packet of the session coming forth. See Figure 7.1.

In the event that we have a better heuristic for determining the start and end of a UDP session (e.g., UDP packets containing NFS requests between the MOUNT and UNMOUNT transactions), the sniffer should use the better heuristic.

Each event report consists of a collection of attributes. Each attribute has a name and a value. Attributes may include the name of the source host, the IP address of the source host, the port number, and the time. All sniffer reports are edge reports.

7.3.1 TELNET

A TELNET connection has several stages (as shown in Figure 7.1). The stages are START, OPTION-NEGOTIATION, AUTHENTICATION, DATA, END or RESET. Each stage is reported by the sniffer with a connection report. We describe the attributes included with each report.

(TELNET,TCP,END,SUCC) refers to the event that FIN packets were observed in both directions. (TELNET,TCP,END,ERROR) indicates that a TCP RST flag was sent in one direction. The other side of the TCP connection may continue to send data and delay acknowledgement. In any case, the sniffer reports the first RST sent in either direction without waiting for its corresponding ACK message. In event of a MOUNT connection, there are attributes that describe the user credential type (e.g., AUTH_NONE, AUTH_UNIX, AUTH_DES, AUTH_KERB), the user credential (e.g., AUTH_UNIX and the path of the filesystem to be mounted. (TELNET,TCP,EXISTING,SUCC) refers to the event that packets of a connection are observed and the sniffer has not seen the start stage of the connection. It may happen when a sniffer is first started. A control variable, ReportPartial, is used to control whether to report this type of events.

The reports are formatted in the DOT-like graph language (see Chapter 3). Here is a report of a START event:

```
digraph sniffer {
   helvellyn.cs.ucdavis.edu -> jaya.cs.ucdavis.edu
     [ app-prot="telnet", prot="tcp", sport=1024,
dport=23, stime =33311222, seq=12345,
stage="start", status="succ"]; }
```

Here is a report of a END event:

```
digraph sniffer {
   helvellyn.cs.ucdavis.edu -> jaya.cs.ucdavis.edu
     [ app-prot="telnet", prot="tcp", sport=1024,
dport=23, stime =33311222, seq=12345,
stage="end", status="succ"]; }
```

Here is a report of a AUTH event:

```
digraph sniffer {
   helvellyn.cs.ucdavis.edu -> jaya.cs.ucdavis.edu
     [ app-prot="telnet", prot="tcp", sport=1024,
dport=23, stime =33311222, seq=12345,
stage="auth", status="error", login="stanifor",
password="fuzbuzz" ]; }
```

7.3.2 NFS

NFS servers and clients use three protocols — PORTMAPPER (RPCBIND), MOUNT, and NFS — to access remote files. An NFS session may consist of transactions to locate the mount server (PORTMAPPER), to mount a file system (MOUNT), and to access files within a file system (NFS). The sniffer only generates reports for selected NFS transactions. The report is sent to the aggregator only after both request and reply messages are processed. The transactions that we process are listed in Figure 7.2.

(NFS,*, AUTH,ERR) reports an RPC protocol authentication failure. The user credential is also included in the report. For example, if a request to access a file is forbidden, the AUTH report includes the user credential "unix/1263/10/10" which indicates that the AUTH_UNIX authentication scheme with user identifier 1263, group 10, and groups 10 was the credential associated with the failed transaction.

type	prot	stage	status	other attributes
NFS	*	AUTH	ERR	user credential
NFS	*	STALE	ERR	file handle
NFS	*	MOUNT	SUCC	path, user credential
NFS	*	MOUNT	ERR	path, user credential
NFS	*	SETUID	ERR	file handle, mode, user credential
NFS	*	READ	SUCC	filename, file handle, user credential
NFS	*	WRITE	SUCC	filename, file handle, user credential

Table 7.2: NFS events.

Since the mount server process (daemon) may be located at a different port address for each host, the sniffer employs a heuristic to recognize mount protocol packets. If first few words of a packet matches the RPC header corresponding to a mount protocol procedure, the rest of packet is processed.

Similarly, the sniffer uses a heuristic to detect the access of interesting filenames by processing NFS lookup transactions and remembering the file handles to those files. Subsequent read, write or setattr transactions on those file handles are reported. For example, the heuristic would treat any file named "passwd" as equivalent and report any accesses to it.

7.4 Sniffer Control Messages

The sniffer will accept control messages to 1) add/change type of packets to sniff 2) ask what the sniffer is currently sniffing for 3) start up and shut down.

The control variables for sniffers are as follows:

startup: To startup a sniffer, set "startup" to TRUE.

shutdown: To startup a sniffer, set "shutdown" to TRUE.

add_endpoint_filter: "add_endpoint_filter" specifies a particular type of connection to be reported by sniffer. It is a list of 3-tuple. Tuples are separated by "
n". Each tuple has three fields: source IP address, destination IP address, and connection type. The fields are separated by whitespace characters. The IP addresses are in full IP address, e.g., rainier.cs.ucdavis.edu. The connection types include: telnet, nfs, www, rip, and rsh. For example, if we are only interested in sniffing telnet connection from k6 to lhotse, and rsh connection from lhotse to nob, then the corresponding tuples are as follows,

k6.cs.ucdavis.edu lhotse.cs.ucdavis.edu telnet
n lhotse.cs.ucdavis.edu nob.cs.ucdavis.edu rsh

Note that if we want to sniffe connection between two hosts, regardless of the direction of the connection, then we need to explicitly specifies the two possible directions by two tuples. For example, the tuples for specifying telnet connections between rainier and k2 are,

rainier.cs.ucdavis.edu k2.cs.ucdavis.edu telnet
n k2.cs.ucdavis.edu rainier.cs.ucdavis.edu

Wildcard, "*", is allowed to represent any IP address or connection type. For example, to sniffer any connection from rainier to sierra,

rainier.cs.ucdavis.edu sierra.cs.ucdavis.edu *

To sniffer any connection, the tuple will be,

* * *

We call a list of these tuples the "wanted-connection".

delete_endpoint_filter:

Delete_endpoint_filter is a list of tuples separated by "
n". Each tuple specifies the connection to be removed from the "wanted_connection". The tuple to be removed must be of exactly same form of as it is added using the "add_endpoint_filter" control variable.

current_endpoint_filter:

It specifies a whole new list for the "wanted-connection". The existing "wanted-connected" list is replaced by this new list.

add_unwanted_connection:

It specifies the connections that are not to be reported. It is also a list of tuple separated by "
n", and each tuple takes the same form as those in "add_endpoint_filter" control variable. We called a list of such tuples as the "unwanted-connection".

delete_unwanted_connection:

It is a list of tuples to be removed from the "unwanted-connection".

current_neg_endpoint_filter: It specifies a whole new list for the "unwanted-connection".

set_time_windows:

It specifies the time of the day to send out reports from the sniffer. It is a list of tuples separated by "
n". Each tuple has two fields: the start time of a period, and the end time of the period. For example, to have report from sniffer between 9pm to noon, and from 1pm to 5pm, set the control variable as follows,

9:00 12:00
n 13:00 17:00

no wildcard is allowed.

set_session_window:

It specifies the maximum length of a UDP session. Nfs reports that are with the same source IP addresses and destination IP addresses, and that fall within the length are combined and reported as a single report.

set_session_gap:

It specifies minimum gap between two UDP sessions. If any two nfs reports that are with the same source IP addresses and destination IP addresses, and that are apart from each other more than the length of session gap, these two reports are regarded as two separate sessions.

7.4.1 Start Up and Shut Down

Everything in this section is a lie. It is controlled by a lying control variable. The variable may be set to "other truth," whereupon the sniffer will report existing connections.

First, when a sniffer is shut down (i.e., no sniffer process it running), control messages from the module controller are silently ignored and no reports are issued.

One can start-up a sniffer by sending a control message to the module controller. To avoid reporting TCP connections and UDP sessions in progress, a sniffer uses the following strategies. The sniffer waits for packets that indicate start of TCP connections. TCP packets that belong to connections already in progress are ignored. Similarly, UDP packets are ignored for α seconds after the sniffer is started. Once a window of quiet time has passed, UDP packets are recognized as UDP sessions (as per the UDP session definition above).

When a sniffer shuts down gracefully (i.e., a control message from the module controller instructs it to shut down), the sniffer sends reports for each TCP or UDP connection that it is currently monitoring and that has not shut down (i.e., FIN packet or timeout window). The report states that there will not be any more reports for this connection due to a shutdown of the data source, not because the connection abruptly terminates. This message tells aggregator rulesets not to expect any more reports and enables each ruleset to clean up appropriately.

When a sniffer shuts down due to an internal fault (out of memory etc.), it should try to send a shut down message to both the module controller and to its aggregator. However, in the case of an unexpected termination of the sniffer, it will not send any messages and the module controller may have to determine that it is no longer functioning and re-activate another sniffer as necessary.

Other alternatives considered but rejected are a fault-tolerant sniffer design that checkpoints its list of monitored connections so that a subsequent invocation can continue to report on existing network connections. This approach was rejected as too complex for our prototype IDS.

7.5 Glossary

connection identifier: a unique id for each connection. The attributes used to construct the connection identifier depends on whether it is a TCP or a UDP connection.

connection report: report of one event.

event: abstraction of a network occurrence.

transaction: a pair of network messages.

7.6 Suggestions to the Implementators

Tcpdump reports one packet per line of output. Upon reading each line, the manager attempts to associate the packet seen with an existing TCP or UDP connection. If unsuccessful, the corresponding packet is dropped. Any information contained in the packet report (the line from tcpdump) that is not implicit in the connection name is formed into attributes for this connection. The connection identifier is formed according to the outline in the communications protocol section.

When the sniffer is started (i.e., the manager process is started), the manager reads a configuration file that describes what connections/ packets it needs to report. (For instance, a sniffer will report all TCP packets that have certain flags–SYN, FIN, RST.) and the expression therefore empty, tcpdump will report all packets.

Sniffers based on SunOS 4.x Network Interface Tap (NIT) cannot monitor packets sent to or from its own interface.

7.6.1 Tcpdump Argument "Expression"

Tcpdump uses a filter "expression" to select packets that satisfy the expression. The absence of an expression implies that all packets are selected. The expression consists of one or more primitives. Primitives usually consist of an id (name or number) preceded by one or more qualifiers. There are three different qualifiers:

Type qualifiers say what kind of thing the id name or number refers to. Possible types are host, net and port. E.g., 'host foo', 'net 128.3', 'port 20'. If there is no type qualifier, host is assumed.

Dir qualifiers specify a particular transfer direction to and/or from id. Possible directions are src, dst, src or dst and src and dst. E.g., 'src foo', 'dst net 128.3', 'src or dst port ftp-data'. If there is no dir qualifier, src or dst is assumed.

Proto qualifiers restrict the match to a particular protocol. Possible protos are: ether, fddi, ip, arp, rarp, decnet, lat, moprc, mopdl, tcp and udp. E.g., 'ether src foo', 'arp net 128.3', 'tcp port 21'. If there is no proto qualifier, all protocols consistent with the type are assumed. E.g., 'src foo' means '(ip or arp or rarp) src foo' (except the latter is not legal syntax), 'net bar' means '(ip or arp or rarp) net bar' and 'port 53' means '(tcp or udp) port 53'.

In addition to the above, there are some special primitive keywords that don't follow the pattern: gateway, broadcast, less, greater and arithmetic expressions. More complex filter expressions are built up by using the words and, or and not to combine primitives.

7.6.2 Tcpdump Example Output

```
Tcpdump output when run without ''expressions''
looks like:
   11:34:35.932301 blanc.cs.ucdavis.edu.43886 >
     rainier.cs.ucdavis.edu.660: udp 56 (DF)

   11:34:36.194128 roma-cafe.cs.ucdavis.edu.1009 >
     avalon.cs.ucdavis.edu.2049: P
     2899044920:2899045044(124)
     ack 1297860147 win 8760 (DF)
```

Tcpdump output in verbose mode looks like:

```
   11:38:32.653805 blanc.cs.ucdavis.edu.43896 >
     rainier.cs.ucdavis.edu.660: udp 88 (DF)
     (ttl 255, id 9012)
```

Tcpdump output when looking for TCP connections looks like:

```
   11:40:24.632285 denali.cs.ucdavis.edu.1023 >
     blanc.cs.ucdavis.edu.login:
     . ack 1873081272 win 4096 (ttl 60, id 31897)

   11:40:24.831506 denali.cs.ucdavis.edu.1023 >
     blanc.cs.ucdavis.edu.login:
     . ack 21 win 4096 (ttl 60, id 31898)
```

Tcpdump output when run with "-e -s96 -S -vv tcp" as expression looks like:

```
   11:52:02.440734 8:0:20:d:f3:52
     8:0:20:23:71:52 ip 60:
     denali.cs.ucdavis.edu.1023 >
     blanc.cs.ucdavis.edu.login:
     . ack 1873104562 win 4096
     (ttl 60, id 33132)

   11:52:02.441077 8:0:20:23:71:52
     8:0:20:d:f3:52 ip 74:
     blanc.cs.ucdavis.edu.login >
     denali.cs.ucdavis.edu.1023:
     P 1873104562:1873104582(20)
     ack 1203776269 win 8760
     (DF) (ttl 255, id 25855)
```

7.6.3 Implementation Plan

1. Create a dummy data-source that takes packet data from a file (e.g., a snoop-format file) and sends the packet data through the communications channel to an aggregator.

2. Since data source library is not finished, perhaps use RPC to perform control.

Chapter 8

Network Access Policies

The purpose of the policy language is to allow a user to specify authorized and unauthorized behavior on the network. A network is a collection of users, hosts and departments. These entities communicate via pair-wise network connections which are labelled with the application protocol employed (e.g., TELNET, NFS, HTTP). Thus, a connection originates from a user, host or department and terminates in another user, host and/or department.

The authorization model employed is similar to an access control model. The user specifies whether a connection is permitted or prohibited. Thus a rule regarding a certain type of connection consists of a tuple (*action*, *time*, *source*, *destination*, *protocol*, *stage*, *status*, ...) where *action* is allow or deny, *time* qualifies the rule with respect to a clock or time interval, *source*, and *destination* describe the connection endpoints and *protocol* describes the connection type. A connection progresses through several stages (e.g. start, login, authentication, stop, etc.), and the *stage* and *status* attribute further characterizes the connection.

The connection endpoints may be described with a user name, a host name and a department name or any combination of the three.

The application protocol is described using by a type (the names of protocol types should be compatible with the protocol names used in the sniffer reports (or other data source reports).

Some protocols may be characterized with additional attributes, for example, connections report for the HTTP protocol may include a URL attribute. The user may specify these additional attributes within each rule if the protocol attribute is not a wildcard.

The policy is translated into a ruleset which is included into the rulesets of those departments that are affected by the policy. The departmental engines interpret the same ruleset based on the contextual information available to it. We illustrate this by an example. Consider the following policy, "no user at host H_1 in department D_1 can telnet to host H_2 in department D_2 and run the program P there", where the common ancestor of department D_1 and D_2, say D_0, is higher up in the organizational hierarchy than both D_1 and D_2. The policy will be translated into a ruleset R which is added to the rulesets of department D_0 and its descending departments. Suppose the sniffer in department D_1 reports to the departmental aggregator a rlogin connection from host H_1 to host H_2. The engine of department D_1 tries to evaluate R against the connection information. However, the engine cannot determine if this rlogin connection violates R, as the engine knows that it will not have the information about the user activities in host H_2. Hence, the engine passes up this information to its parent aggregator, hoping that the required information will be available at a higher level in the structure. Similarly, any department between D_0 and D_1 in the hierarchy structure (if exist) will pass the rlogin connection information to its parent. When the rlogin connection information arrives at the aggregator of department D_0, the engine knows that all the required information for evaluating R can be available at this level, and it will make the decision if the connection violates R.

In order for this scheme to work, the engine of a department needs to have the knowlege about the set of information available to its departmental aggregator which is a function of the detection modules reside in the department.

One issue that must be addressed further is how to resolve conflicts between rules. For example, the user may specify a pair of rules, the first authorizes a type of connection and the second prohibits it. The syntax of the policy language cannot prevent this. Thus, the compiler should detect and warn the user when policies contain conflicting rules.

The rule (deny, *, A/D1, D2, TELNET, AUTH, SUCC) generates the following ruleset:

```
#-----------------------------------------------
# User A in dept D1 cannot telnet to dept D2

node precondition new.source.dept == 'D1'
     && new.dest.dept == 'D2';
edge precondition new.edge.app-prot == 'telnet'
     && new.edge.suser == 'A';


node rules {
  res.node.combine = 1;
}

edge rules {
  res.edge.combine = 1;
}

assessments {
  global.nedges >= 1 ==> alert, report-graph;
}
```

The rule (deny, *, α, α, *) generates the following ruleset, where α represents an instantiated variable. The variable α is instantiated with the value of the source and destination attributes and both must match to trigger the rule.

```
#-----------------------------------------
# User A can only make connections within
# a single department

node precondition new.source.dept == new.dest.dept;
edge precondition new.edge.suser == 'A';


node rules {
  res.node.combine = 1;
}

edge rules {
  res.edge.combine = 1;
}

assessments {
  global.nedges >= 1 ==> alert, report-graph;
}
```

The rule (deny, *, *, *, NFS, READ, SUCC, filename(passwd)) generates the following rulesets.

```
#---------------------------------------------
# report movement of passwd between hosts
# using NFS

node precondition 1;
edge precondition
     in_set('passwd',new.edge.files_moved)
     && new.edge.app-prot == 'nfs');

node rules {
  res.node.combine = 1;
}

edge rules {
  res.edge.combine = 1;
}

assessments {
  global.nedges >= 1 ==> alert, report-graph;
}
```

Our current policy language is not able to specify the following ruleset. However, the language must be extend to allow such a ruleset to be generated.

```
#--------------------------------------------
# a single user using a series of connections
# is suspicious

node precondition defined(new.node.name);
edge precondition new.edge.connection;

node rules {
  res.node.combine = !defined(cur.global.User)
  || new.global.User == cur.global.User;
}

edge rules {
  res.edge.combine = 0;
  res.global.User= new.source.suser;
}

assessments {
  global.nedges >= 4 ==> alert, report-graph;
}
```

8.0.4 Language Syntax

The syntax of the policy language is not yet specified. The policy language syntax should allow multiple rules, variable number of attributes in the rule, the specification of instantiated variables, wildcards, the specification of combinations of users, hosts, and departments, and attribute values that are lists.

Chapter 9

Debugging Facilities

9.1 Overview

Debugging capabilities should be integral to the design of all relevant GrIDS components. Two types of debugging capabilities are supported. The first type is for providing a logging facility for GrIDS components. Using this logging service, a GrIDS component can record debugging messages in a central place. The second type is for debugging rulesets. Using this ruleset debugging service, a rule writer can understand how the rulesets operate among GrIDS components.

9.2 Central logging facility

The central logging facility is a simple means for GrIDS components to record debugging messages in a central place. The facility is expected to be deployed during the internal development phase of GrIDS. We will not ship (at least we will not support) the implementation of this type of debugging capabilities as a part of GrIDS.

When a component detects an exception, it can log the event using the central logging facility. Thus instead of using "die" or "warn" Perl statements, GrIDS module writers should use the central logging facility to record exceptions.

When a GrIDS component wants to log a message, it will invoke a library routine **grids_log** with the string it wants to log. The centralized log will record the event with the source and the local timestamp. The centralized log thus induces a partial ordering of the recorded events. The central logging facility can be implemented using NFS.

We illustrate the use of grids_log with an example:

```
#IF DBUG
grids_log("before send ctrl msg");
#ENDIF
send_con(...);
#IF DBUG
grids_log("after send ctrl msg");
#ENDIF
```

Note that if an Engine discovers a syntax error in a Ruleset, that message should be sent to the central logging facility.

9.3 Ruleset debugging servers

The purpose of a ruleset debugging server or simply a debugging server (DBG) is to respond to requests (originally) from a user interface (typically off-host), for sections of debugging logs stored on-host.

Assuming logging of the relevant information is enabled for a module (via one or more dynamically settable State Vars), this will allow a retroactive reconstruction/tracing of distributed processing in GrIDS.

In order to reduce the number of processes, ruleset debugging servers are not implemented by a dedicated process. Instead, the debugging servers are implemented as part of Module Controllers.

To debug Rulesets, a debugging server running on each host will respond to requests (directly from a Software Manager; indirectly from a User Interface) to send specific debugging information regarding the behaviors of aggregators. That information will be displayed at the User Interface without much additional processing. As we begin using GrIDS, it should become clear which enhancements to the debugging library routines at the User Interface end should have the highest priority.

The DBG waits for a TCP connection request responds to the request, then waits for that party to send another request, or for a new connection from another party. It is the responsibility of the client party to close a DBG TCP connection.

9.4 Log browsing

DBG supports three types of services: log browsing, forward data flow tracing, and backward data flow tracing. Data flow tracing requires the user to specify a unique label or connection ID for the edge to be traced. Because we have no way to label **subgraphs**, they are accessible only in browsing mode.

Browsing allows a wide variety of info emitted by Engines to be viewed. (For details on what Ruleset debugging info the Engine emits, see section 2.8.)

Log browsing provides a means to retrieve a certain portion of an aggregator log that satisfies given criteria. The other two services will be described subsequently. There are three types of messages used for browsing aggregator logs.

Log Browsing Request: This message represents a request to a debugging server to collect reports that fall into a specified time interval and belong to a specified aggregator. The message has four terms: start-time, end-time, department ID, and *optional* Ruleset Name. Start-time and end-time are ASCII representation of Unix time stamps. There are two special values for the time stamps. 0 means the earliest possible time, and -1 means the latest possible time. Department ID is the unique label assigned to a department.

Start time	*End time*	*Dept ID*	*Ruleset Name*

Log Browsing Result: This message is sent in response to a Log Browsing Request. It supplies all the reports that were sent to the aggregator within the specified time interval. The format of the reports is shown in Section 2.8. The message has four terms (plus optional Ruleset Name): start-time, end-time, department ID, and report-list. The first four terms are copied from the corresponding request. Report-list is a list of reports.

Start time	*End time*	*Dept ID*	*Ruleset Name*	*Report list*

Log Browsing Error: This message is sent in response to a Log Browsing Request and indicates that the request cannot be successfully processed. The message has four terms (plus optional Ruleset Name): start-time, end-time, department ID, and error. Error is a string explaining why an error occurred. Typical reasons are that data requested not available (e.g., data for time specified not available), and unknown department ID.

Start time	*End time*	*Dept ID*	*Ruleset Name*	*Error*

9.5 Serving log browsing requests

When a DBG receives a Log Browsing Request, how does the DBG translate that info into (an) *exact* pathname(s)?

This question has been answered by clever design of the Module Controller. We specify that every host's Module Controller must maintain a "*Current Task File*", in which it stores various data about each living GrIDS module it spawned. The Module Controller needs this info in persistent storage, so that whenever it starts up, it can "adopt" any still-living children it spawned in a past life. For each child process, the "*Current Task File*" will contain:

- *PID* of the child (so the Module Controller can *signal* the child when it receives a SET command for it).
- name (*generic alias*) of the child GrIDS module.
- *Department_ID* of that child module (since there may be several identical modules running on a host, representing different Departments).
- *pathname* of config files for this child. This might indicate a separate *subdirectory* for each child's files, or it might be a pathname *prefix* unique to one child. In either case, the implementation team must devise some standard *naming conventions* – probably using suffixes – to specify the various config-files for one GrIDS module. Config files for a module include *initialization* file, *command* file, *status/response* file, and *debug/logging* file(s).

Thus, when a DBG receives a Log Browsing Request, it can find the exact location of the desired debugging logfile by reading the Module Controller's *Current Task File*.

Note that the Log Browsing Request might imply *multiple* logfiles. In the most extreme case, an Aggregator for Department Physics might have *multiple* logfiles that store debugging information – one logfile per Ruleset per hour.

9.6 Control variables

Control variables currently envisioned for each DBG are as follows:

- exact *pathname* of Debugger's *Current Task File*. (By resetting this to an *archival Current Task File*, a UI could direct the DBG to peruse *post – mortem* logfiles of GrIDS modules that are no longer alive.)

9.7 Access Control

For uniformity of access control, we decided a User Interface must contact a Software Manager with a debugging request. That Software Manager will check *human* access control, then forward the request to the proper Debugger (Module Controller).

Once a Debugger receives a request, it will follow the normal *machine* access-control procedure it uses in its role as Module Controller. For example, prior to opening the logfile for a "resident-alien" Aggregator, the DBG should first read the Aggregator's own ACL, to ensure the Host:Port of the requesting Software Manager matches that of the foreign Aggregator's home department (vs. the Host:Port of the Debugger's own home department).

(See *access – control* discussion in software_control.tex)

9.8 Requirements for others

Note to implementors of Engine, Sniffer, Protocols, and User Interface :
For the debugging routines to achieve forward and backward dataflow tracing of graph pieces, the UI will need some means to uniquely identify or label graph pieces. Those labels must be preserved as edges become "reduced" during upward aggregation. Also, an Aggregator's Global Log File must note the *source* of any report that comes from a child Aggregator (vs. from an on-host data source).

Note to implementors of Software Management System: We don't want user to have to manually turn on $debug{foo} for 25 departments. So let's allow that control-var to be propagated downward via *inheritance*. Presumably, user should turn on/off $debug{foo} at the highest node at which Ruleset *foo* is defined.

9.9 Forward data flow tracing

9.9.1 Overview

Given a piece of data we want to trace, forward data flow tracing presents to the client how the data got propagated in GrIDS. The type of data we want to trace are events associated with a connection. Examples of connection events can be found in the Network Monitoring chapter.

9.9.2 Specifying the target connection

To use forward data flow tracing, a user needs to specify which connection to focus on. The Communications Protocol chapter specifies the set of attributes used to uniquely identify a connection. However, a user normally does not know all the attributes needed to identify a connection. For example, an attribute used for identifying a connection is the start time observed by the sniffer. Thus having the user to directly specify the ID of the connection does not work.

For the user to locate the ID of the connection, the user first sends a Log Browsing Request (time interval, department ID) to a debugging server. The debugging server then returns the user a list of connection events seen by the aggregator of the corresponding department. A department ID is included because there may be multiple aggregators running on a machine, and the debugging server needs to know which aggregator's log to use for the reply.

9.9.3 Forward-tracing debugging algorithm

The protocol for forward data flow tracing consists of two phases: log browsing and forward tracing. It is permissible to browse an engine log without proceeding to the forward tracing phase.

1. A client sends a Log Browsing Request, (time interval, department ID), to the debugging server on the host of department ID's engine.

2. The debugging server returns a list of event reports that match the department ID (and optional ruleset name) and fall within the time interval. Based on the list, the client may narrow the search by selecting one connection ID to target for forward tracing. Or the client may terminate the search.

3. The client finds out the location of the parent aggregator machine and sends a forward tracing request, (connection ID, time interval, department ID, ruleset name), to the debugging server on that machine. The parent aggregator is now the current aggregator, and "department ID" corresponds to the name of this aggregator's department.

   Again in this case, *ruleset name* is an optional additional filter criterion. If *ruleset name* is not specified, then the debugging server will select event reports for *all ruleset names* in which the specified *connection ID* appeared.

4. The debugging server sends forward tracing results to the client. If the results are "positive" (i.e., the reports passed the pre-qualifying rule of a relevant ruleset), then we assume the ruleset is inherited, and attempt to goto the previous step. Otherwise, exit the protocol.

We envision that a user usually wants to debug rulesets one at a time. Because rulesets are independently executed, one does not need to worry about interference among rulesets. Thus it is always possible to debug rulesets one at a time. However, we allow the simultaneous tracing of one connection through multiple rulesets, since the user may wish to compare the behavior of two or more slightly different rulesets.

The above protocol only describes the "normal" behaviors of the protocol. There are cases in which the above protocol might not work. We assume the following:

- The organizational hierarchy has not been changed between the time of logging and the time of running this protocol.

- Logging is turned on for the specified rulesets on the aggregators involved.

- The attributes of the connection ID are kept as the data propagate in GrIDS.

- The relevant hosts are up.

9.9.4 Forward tracing messages

There are six types of messages used in the above protocol. The three of them that concern log browsing have already been described in the Log Browsing section. The following three are for forward tracing.

Forward Tracing Request: This message represents a request to collect reports about a connection within a specified time interval that optionally relates to a ruleset. The message has the following terms: <connection ID>, start-time, end-time, department ID, and optionally ruleset name. The attributes used to label a connection, <connection ID>, are shown in the Communications Protocol chapter.

Conn ID	*Start time*	*End time*
Dept ID	*Ruleset Name*	

Forward Tracing Result: This message is sent in response to a forward tracing request. For each report that matches the conditions specified in the request, a boolean result showing whether the report passed the pre-qualifying rule is also returned. In addition to the terms copied from the request message, the message includes a list of reports that match the conditions prepended by the ruleset name and the result of the pre-qualifying rule test.

Conn ID	*Start time*	*End time*
Dept ID	*Ruleset Name*	*Report list*

Forward Tracing Error: This error message represents the forward tracing request received cannot be processed. The message has the following terms: <connection ID>, start-time, end-time, department ID, optionally ruleset name, and error. Error explains why an error occurred. Typical reasons are that data requested not available, incorrect department ID, incorrect ruleset name.

Conn ID	*Start time*	*End time*
Dept ID	*Ruleset Name*	*Error*

9.10 Backward data flow tracing

9.10.1 Overview

The protocol for backward data flow tracing consists of two phases: selecting a connection, and backward tracing. Selection may occur as a result of browsing an engine log at an intermediate aggregator (as described in the section on Log Browsing), or it may occur from visual inspection of a graph that has been displayed at a user interface for whatever reason.

In any case, we assume the user has selected a specific connection, and knows its *Connection ID*, the *Ruleset Name* in whose graph space it appeared, and the *Department ID* of the aggregator at which to initiate the trace.

9.10.2 Backward-tracing debugging algorithm

1. The client finds out the location of the aggregator machine and sends a backward tracing request, (connection ID, time interval, department ID, ruleset name), to the debugging server on that machine.

Conn ID	*Start time*	*End time*
Dept ID	*Ruleset Name*	

2. The debugging server returns to the client a list of event reports that match the selection criteria and fall within the time interval. (Typically, only a single event would qualify within the time interval.)

Conn ID	*Start time*	*End time*
Dept ID	*Ruleset Name*	*Report list*

3. The client examines the event report(s) to determine its *source*. If the *source* is not a terminal hostname, but rather a sub-department ID, then goto step one. Otherwise, exit the protocol.

Again, the above protocol only describes the "normal" behaviors of the protocol. There are cases in which the above protocol might not work. We assume the following:

- The organizational hierarchy has not been changed between the time of logging and the time of running this protocol.
- Logging is turned on for the specified rulesets on the aggregators involved.
- The attributes of the connection ID are kept as the data propagate in GrIDS.
- The relevant hosts are up.

Chapter 10

The User Interface

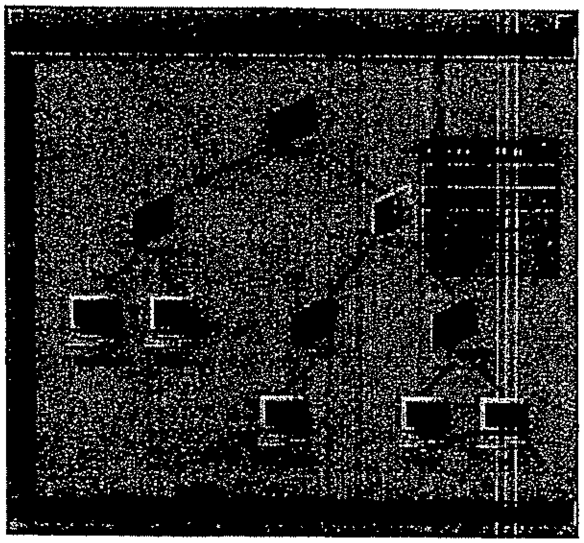


Figure 10.1: Hierarchy window after clicking on CS department icon.

10.1 Logging In

On startup, the user interface will promt the user for a username, a password and a department to administer. This information will be used to obtain a copy of the hierarchy below the requested department as described in chapter 6. Only after this information has been obtained will the user move beyond the login prompt into the interface proper.

Note that it is not possible in the current version of GrIDS to administer several departments at the same time except by administering a common ancestor of all of them.

10.2 Managing the hierarchy

The interface will present to the user a window which shows the organizational hierarchy. Departments and hosts will be shown as appropriate icons, and the hierarchical relationship between them will be displayed as a tree, as shown in Figure 10.1.

All operations on the hierarchy will be achieved by first selecting a node by clicking on it via a mouse, and then by issuing an appropriate command through a menu or through a mnemonic keyboard shortcut. This will bring up an appropriate dialog panel in which additional information necessary to the transaction will be entered. The interface will then initiate the appropriate hierarchy transaction.

We now go through the possible actions on a node and give details of the transactions that can be performed. In each case, the transaction will be perfomed on the hierarchy. Only when it is complete will the user's display be updated. Should the transaction fail, an error panel will be put up informing the user of the problem.

10.2.1 Add Host

Selected:	A Department
Dialog parameters:	Name of new host Port of module controller

The new host is created below the selected department. Note that a module controller must already have been started on the host in question, and its port must be supplied manually to the system. (After this, everything can be handled automatically, but we need this initial manual input to get a foothold on the system).

10.2.2 Add Dept

Selected:	A Department
Dialog parameters:	Name of new department Host for aggregator. Host for software manager.

The new department is created as a child of the selected department. The user must make the decision as to where the aggregator and the software manager will be located, since this decision is likely to depend on factors not available to the system (performance and security of machines in question)).

10.2.3 More transactions

There will be a whole bunch more, deriving in a natural way from the ones listed in chapter 6, but they will be added here when implementation of them is a little closer.

10.3 Managing rulesets

Upon selecting a department node in the hierarchy view and giving the appropriate menu option, the names of all rulesets running on that aggregator will be presented. Selecting any particular ruleset name causes the graphs in that graph space to be drawn. Two kinds of operations are supported on nodes by selecting them and giving the appropriate command. For departments only, it is possible to expand the node to the corresponding graph in the department which this node represents. For both host and departments, an attribute panel can be brought up which shows all the attributes of the node in this particular graph. A button or equivalent should be provided to move upwards in the aggregation scheme.

When any department is selected, the option of making queries to the corresponding aggregator is provided. The query will be entered as text, in some text editor (of the implementors' choice), and fed to the engine. The query should be in the query language, as specified.

When any department is selected, with all ruleset names displayed, the option of viewing the ruleset text is provided as well as viewing the graphs associated with that set. While viewing the text, modifications may be made to the text, and an entry button clicked on when finished to ship the new version of the ruleset off to the aggregator. When all ruleset names are displayed, the option of creating another ruleset (by entering text) is also provided.

10.4 Alerts

A seperate window is maintained for alerts. In this window, an icon for each ruleset is shown, with icons containing unviewed alerts in yellow or red (depending on the severity of the alert) as shown in Figure 10.2. Clicking on the "Sweep" icon generates the popup window shown in Figure 10.3. This window displays the most recent alert, showing the text of the alert above the graph of the alert (both of which are sent by the graph engine automatically when alerts are generated.) Selecting from the alert history bar, one can see previous alerts (see Figure 10.4).

10.5 Managing Software

GrIDS provides features so that the IDS itself can be managed conveniently. Here we describe the interface to those features.

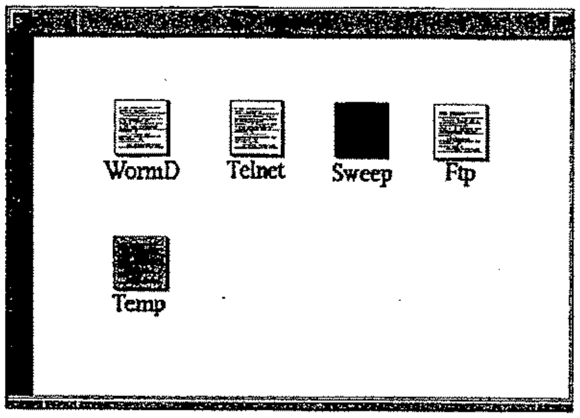


Figure 10.2: Main alert window displaying icons for each rule set. Rule sets which have generated alerts are shown in yellow and red.

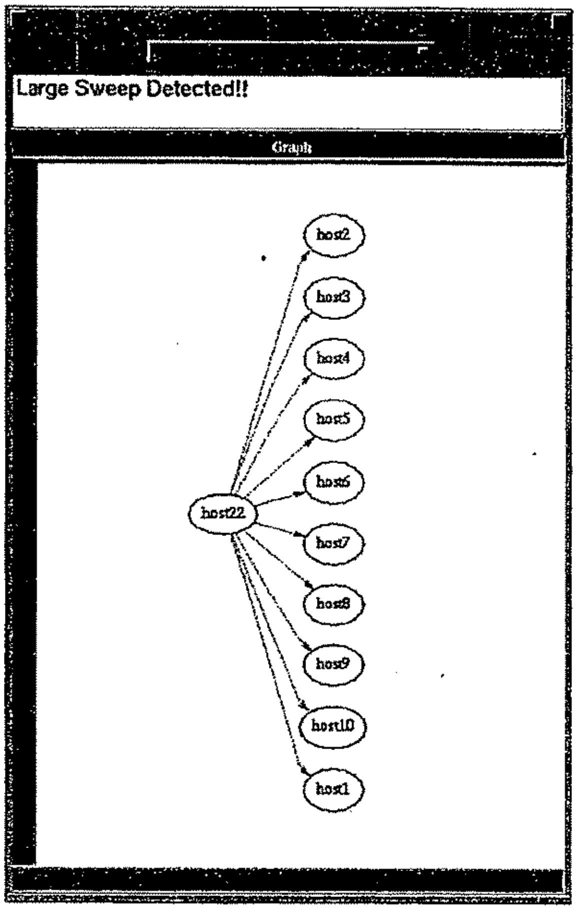

Large Sweep Detected!!

Figure 10.3: Window resulting from clicking on Sweep in main alert window.

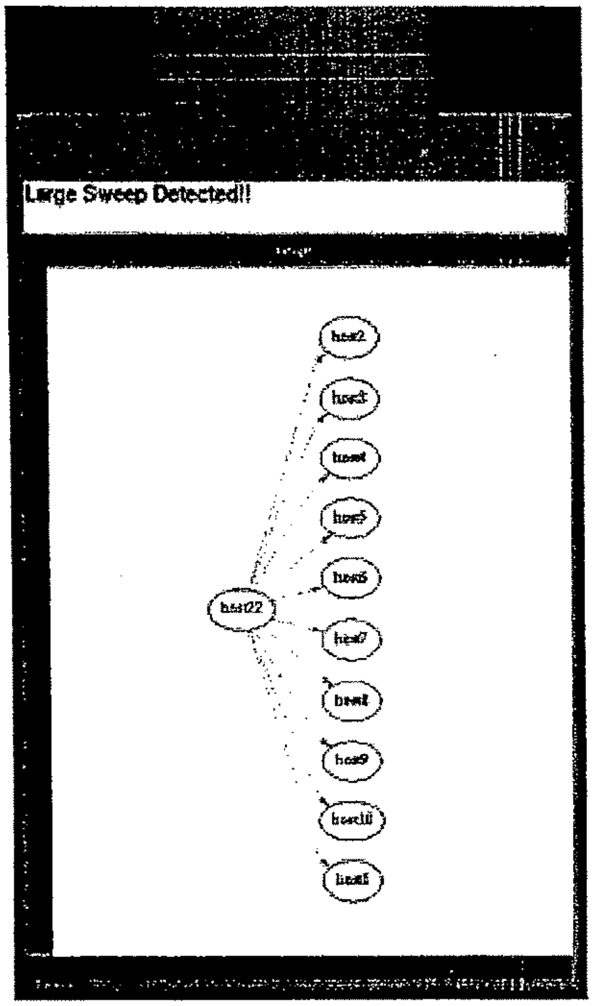


Figure 10.4: Window after clicking on history bar.

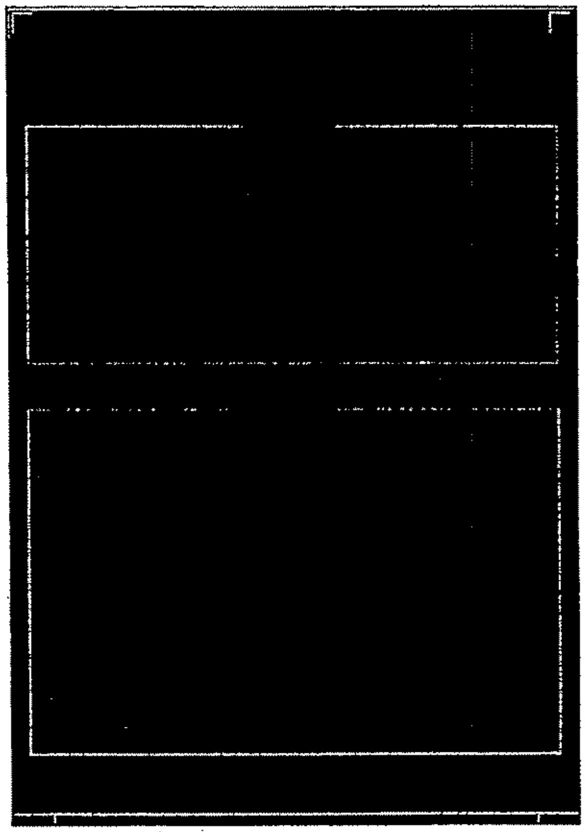


Figure 10.5: Inspecting the GrIDS system for a department.

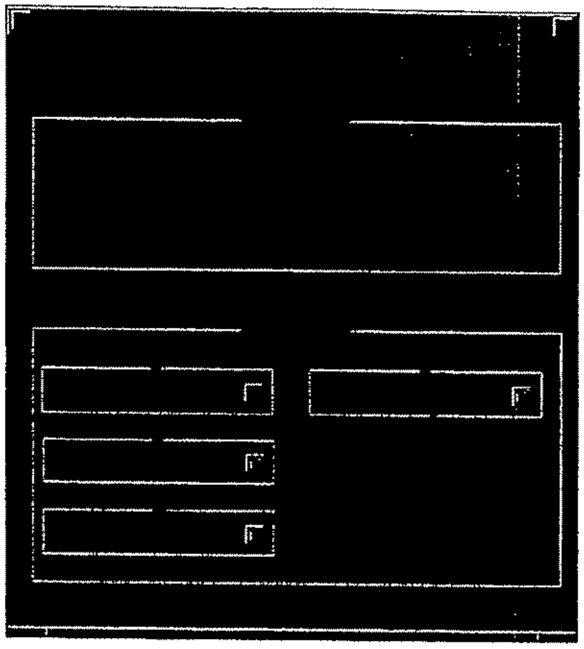
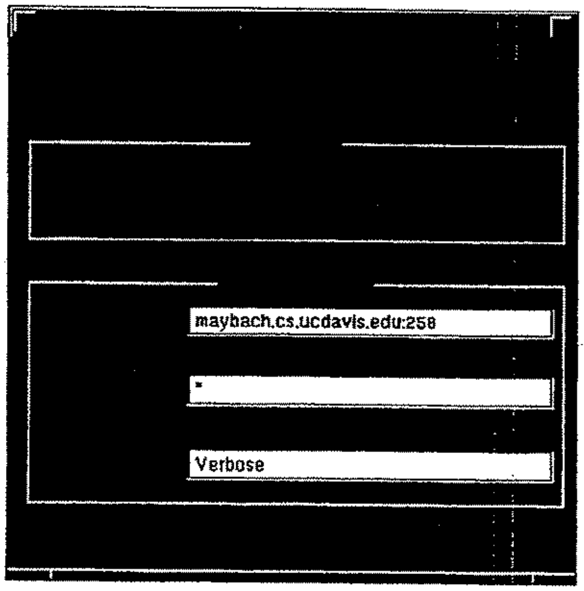


Figure 10.6: Inspecting the GrIDS system on a particular host.



maybach.cs.ucdavis.edu:258

*

Verbose

Figure 10.7: Inspecting a particular module.

10.5.1 Department Inspector

From the hierarchy view, after selecting a particular department, a department inspector panel can be brought up. This shows summary information about the department (such as where its infrastructure is located), and provides statistics on what kind of data sources are running at or below this department.

10.5.2 Host Inspector

From the hierarchy view, after selecting a particular host, a host inspector panel can be brought up. This provides a view similar to that shown in Figure 10.6. At the top, the panel provides a summary of the GrIDS infrastructure components running on the host (in this case, the module controller and two aggregators). These cannot be manipulated directly. Below that are shown possible data source modules, together with switches which allow them to be started or stopped.

10.5.3 Module Inspector

From the host inspector or the department inspector, it is possible to bring up the module inspector (shown in Figure 10.7). This provides an overview of the status of a module, and allows the user to alter control variables. Control variables which should not be directly manipulated (here the aggregator to which this data source responds) are shown greyed out. Some control variables may have extensive text. In this case the text field should be replaced with a button to bring up an extra edit window for viewing and altering the variable.

Appendix A

Tracing Using GrIDS

A.1 Introduction

This appendix covers the issues involved with tracing intruders using GrIDS. Here *tracing* means finding out who is responsible for a particular activity on a network, even though they may have taken various steps to disguise their identity (such as logging in through multiple compromised accounts). The tracing problem has been discussed in more detail elsewhere. [?, ?]

We consider two possible tracing mechanisms. The first we call on-host tracing. Here, the basic primitive available is that some channel into a host and some channel out are, in fact, causally connected. The second is thumbprinting. [?] Here the available primitive is that two network connections, which may or may not be close together in the network topology, have identical content. In each case, we take up the question of how GrIDS can make use of the primitive in question to trace the source of an activity.

A.2 General Considerations

The essence of tracing is that one waits for something to happen (probably something bad), and then says "who did this?"

What GrIDS does is to build graphs incrementally as information arrives. This is inherent in its design. Thus, in order to answer a tracing query, GrIDS must already have built the graphs it needs *before* it receives the query. It is then only capable of modest analysis to determine which graphs are required to answer the query.

Let us take the example of users telneting through multiple machines (possibly using different accounts on each one) before perpetrating some crime. We, detecting the crime, would like to trace back through the sequence of telnets to determine the origin of the criminal activity.

The only approach possible in GrIDS is something like the following. As we receive reports of telnet connections, we form them into graphs with the proviso that all causally related links are in the same graph, while links which are not causally related must be in different graphs. Then, GrIDS must have the ability to search through its graphs looking for ones which contain some particular host or connection of interest, and returning this.

Thus GrIDS must be able to handle queries for graphs in addition to reporting all graphs which cause some rules to fire.[1]

A.3 GrIDS Query Language

The GrIDS engine needs to a have a query mechanism to support tracing. A sufficient mechanism might provide the features to return all graphs which match certain criteria. At a minimum, these criteria should be able to refer to the global attributes of the graph. However, it is essential for tracing that the criteria also refer to local attributes.

For example, typical queries might be (in English), return all graphs which

- have an incoming telnet to host X
- have an incoming telnet to department Y
- involve an NFS mount of filesystem F on server S.
- begin on host X
- involve user U

We restrict ourselves to what is needed for tracing. Many more general queries can be imagined for other purposes.

Thus, a reasonably general query mechanism needs at least the ability to say "give me all graphs which include host X, where that host has attributes which satisfy condition P_X and where the global graph attributes satisfy P_G. Such queries need to be parsed at request time, the resulting graphs selected, and the graphs shipped out to the requester.

A.4 On-Host Tracing

Suppose we have two network channels, C_1 and C_2 which share at least one endpoint. The primitive which an on-host tracing system provides us with is an assertion

$$\text{caused\_by}(C_1, C_2) \qquad \text{(A.1)}$$

[1]The alternative is to pass all graphs to some special purpose module which handles the queries. However, this is very inefficient since it requires updating complete copies of all graphs in two places instead of one.

This indicates that the activities occurring in C_2 are caused by C_1. For example, C_2 may be a sendmail connection which results from mail being sent during a login session C_1. This kind of information can usually only be obtained when adequate instrumentation is present on the host in question.

One example of such an on-host tracing system, *Foxhound*, was described in [?]. That system worked (under Unix) by examining the process table of the host, and associating processes with their parent processes and with the connections they owned. For our purposes, it is of only passing interest how the predicate in equation A.1 is computed. Our concern is with how it can be used by GrIDS.

From GrIDS' perspective, the incoming information cannot be naturally modelled as a link attribute, since it involves two different links. However, since the links share an endpoint, it could be modelled as a node attribute at that shared endpoint. The report would say that node X has the attribute caused_by(C_1, C_2), where C_1 and C_2 would use whatever scheme we finally settle on for referring to links. Note that this idea requires that attributes are not constants but rather link to other things in the graph topology.

The alternative to viewing the report as a node attribute is to say that it is a special kind of report - a "correlated links" report, which gets handled by its own set of rules. We note that the same kind of report is needed in aggregation, where we have to pass up reduced graph nodes and the incoming and outgoing links from them; we need to distinguish whether such links are actually connected or not in the graph.

A.5 Thumbprinting

The primitive in the thumbprinting could be one of several things. In the simplest case, we could just get thumbprints as link attributes which represented the numerical values of the thumbprints. Then we could associate links together into graphs based on whether their thumbprints matched. In the current graph execution model, this would require storing the attributes of the links also as node attributes so that an incoming link could be compared with them.

We would need to import some suitable function which knew how to compare thumbprint attributes. Note that a thumbprint is a somewhat complex piece of data - it is neither a scalar, nor a set, but rather a two dimensional array of scalars (several numbers for each time slice). This suggests that attributes need to have relatively complex types, and that we need to be able to import code for operations on those types.

In the second case the primitive could be something like

$$\text{causally\_linked}(C_1, C_2) \qquad (A.2)$$

where there is no longer any requirement that C_1 and C_2 share an endpoint. This could be a report from some external thumbprinting system which got reports of all thumbprints, sorted them using appropriate spatial data structures, and then reported when any got close together.

Given these reports, we no longer need to import fancy code to compare attributes. We could use these reports exactly as in the previous section (for on-host reports), and just throw away any reports where C_1 and C_2 happen not to share an endpoint. This rather limits the benefits of thumbprinting though. The alternative is to start building disconnected graphs so that we can lump together connections which we know *must* be causally connected, even though we do not know the topology of the connection between them. This opens rather a can of worms.